RJ10531 (ALM1509-001) September 25, 2015
Computer Science

IBM Research Report

Adaptive Caching Algorithms for Big Data Systems

Avrilia Floratou[1], Nimrod Megiddo[1], Navneet Potti[2], Fatma Özcan[1], Uday Kale[3], Jan Schmitz-Hermes[4]

[1]IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

[2] University of Wisconsin-Madison

[3]IBM Analytics Group

[4]IBM Deutschland

Adaptive Caching Algorithms for Big Data Systems

Avrilia Floratou[†], Nimrod Megiddo[†], Navneet Potti[⋆], Fatma Özcan[†], Uday Kale[‡], Jan Schmitz-Hermes[§1]

[†]*IBM Almaden Research Center* {aflorat, megiddo, fozcan}@us.ibm.com

[⋆]*University of Wisconsin-Madison* nav@cs.wisc.edu

[‡]*IBM Analytics Group* udayk@us.ibm.com

[§1]*IBM Deutschland* jan.schmitz-hermes@de.ibm.com

September 29, 2015

Abstract

Today's Big Data platforms have enabled the democratization of data by allowing data sharing among various data processing frameworks and applications that run in the same platform. This data and resource sharing, combined with the fact that most applications tend to access a hot set of the data has led to the development of external, in-memory, distributed caching frameworks. In this paper, we develop online, adaptive algorithms for external caches. Our caching algorithms take into account the workload access pattern, and the cost of insertions in the external caching framework when making cache insertion and replacement decisions. We provide both a detailed simulation study as well as cluster experiments on IBM Big SQL, and show that *only* our adaptive algorithms perform well for different workload characteristics, are able to adapt to evolving workload access patterns, and can approach the performance observed by optimized offline solutions.

1 Introduction

Enterprises are using the Hadoop Distributed File System (HDFS) as their central data repository, storing all their enterprise data, including IoT and mobile applications. The new Big Data platforms, like Hadoop and YARN, enable enterprises to share their data among multiple frameworks. It is common for enterprises to run their SQL applications, machine learning and advanced analytics, graph and streaming analytics in a single platform. Moreover, none of these frameworks own the data, instead they all work on open HDFS data formats, and share the data. This democratization of data in the Big Data platforms, and the need to co-exist with different applications and frameworks have brought new architectural requirements. For example, to exploit larger memories, the current generation of Big Data platforms [12, 30] provide external, distributed caching mechanisms to cache HDFS data in memory. In particular, HDFS caching [13] and Tachyon [21] are two approaches to storing data in memory.

We call these systems, *external caches*. These caches are shared among different applications, and hence are different from the traditional buffer pool mechanisms. Buffer pools store the data in the internal format of the database, whereas the data in HDFS cache is stored in the original file format (e.g, Parquet, Text, ORCFile, Sequence File), and still needs to be converted to the internal representation that is needed by the particular application. As a result, the external caches help reduce I/O costs, but not necessarily CPU costs. Another important difference between external caches and buffer pools is that all data access is carried out through the buffer pool in a database system. Hence, if a page (or an object) is not in the buffer pool, it is first brought there. As a result, most caching algorithms, such as LRU, focus on which pages to evict from the buffer pool. However, a cache miss is handled differently in our setting. First, cache insertions are more costly, because insertions are executed by the process that manages the external cache, such as HDFS cache [13], which competes for resources with the application that needs the data, such as the SQL system. In fact, in our experiments we observed that traditional caching algorithms such as LRU which assume that all data accesses go through the cache, might actually result in worse performance than the performance obtained by completely ignoring the cache and reading the data directly from secondary storage. Second, when needed data is not in the external cache, the application can directly read the data from disk. As a result, which objects to insert into the cache is as important a decision as to which objects to evict from the cache.

The workload access pattern is an important factor to consider when making data caching decisions. Most Big Data applications access only a small percentage of the data [26]. In other words, there is a hot set of data that changes over time, and a given application does not have to access petabytes of data at a time. Thus, if we can figure out the hot set for each application, we can, then, cache that data in an external cache to speed up data processing.

In this paper, we propose algorithms to identify the hot data set to store in an external cache by observing the workload data access pattern. Different applications have different data access patterns. A machine-learning algorithm iterates over the same data set multiple times. For these applications, the user can explicitly pin the data set in memory. However, in many other applications like SQL, graph analysis, and data transformations, data access patterns vary significantly. Two important parameters that determine the data access patterns are frequency and recency of access. For example, an OLAP application may access the same portion of the fact table frequently for a while because the analytics works on a time window. Within this access pattern, it may also access other tables. Hence, the most-recently-accessed data items are not always the same as the most-frequently-accessed ones.

A plethora of algorithms have been developed to address the needs of different applications. These algorithms optimize for various data access patterns. For example, the LRU-K [25] method is a popular buffer pool eviction strategy, whose behavior heavily depends on the recency of data accesses, characterizing the most-recently-accessed data as the hot set. On the other hand, the LFU (Least Frequently Used) method takes the frequency of data accesses into account and does not consider the recency of data accesses when making cache replacement decisions. In this paper, we first adapt the well-known LRU-K algorithm to work with variable size objects and external caches. This new algorithm, SLRU-K, remembers the last K accesses to objects to give some weight to frequency of data accesses. But, we observe that it is not able to capture frequency properly, and emphasizes the recency of data accesses more, resulting in poor cache performance.

To strike a better balance between recency and frequency, we propose a novel algorithm, EXD, which makes use of a single parameter that determines the weight of frequency vs. recency of data accesses. This algorithm also takes into account the cost of a cache miss, and the probability of re-access for each object. The EXD algorithm is based on the knapsack formulation [16] and uses an exponential function to estimate the probability of object accesses. The algorithm can perform very well under various workload access patterns if the parameter is set correctly. However, the correct value of the parameter depends on the application, and our goal is to support various different applications through a single external cache. For this reason, we develop an adaptive method that observes the workload characteristics, and dynamically adjusts the value of the algorithmic parameter. Our adaptive algorithms are able to adapt to changes in workload access patterns, and thus, can accommodate the needs of various applications.

Our contributions can be summarized as follows:

- We develop online caching algorithms (`SLRU-K, EXD`) for external caches, which cache popular objects based on various metrics such as frequency, recency, cost of miss, and probability of re-access.
- The proposed algorithms selectively cache only the most significant objects to reduce the overhead of insertion into the external cache.
- We propose parameter-free, adaptive versions of our caching algorithms (`Adaptive SLRU-K`, `Adaptive EXD`) that are able to adjust to various workload access patterns.
- We extensively evaluate our proposed algorithms, using a simulation study based on three different workload generators, to explore the whole spectrum of data access patterns, and show that our algorithms can adapt to various data access patterns.
- We incorporate our caching framework in Big SQL, IBM's SQL-on-Hadoop offering and evaluate it on a cluster environment using a TPC-DS like benchmark, that has been used by multiple SQL-on-Hadoop vendors, as well as additional synthetic workloads. We show that our adaptive techniques can provide much better performance than existing static algorithms, can approach the performance observed by optimized offline solutions, and produce the best performance for diverse workloads that contain a mix of concurrent batch and interactive queries.

2 Caching Problem Foundations

The task of maximizing the expected performance of a cache has been modeled in literature as a knapsack problem [16, 14]. In this well-known formulation, it is assumed that caching an object provides certain benefit (future accesses to the object will be hits) and the cache policy has to maximize the total expected benefit from the cache given that the total size of the cached objects cannot exceed the size of the cache. Most well-known caching algorithms can be viewed as different solutions to this knapscak problem that differentiate based on the model that they use to estimate the probability of re-accessing an object in the future. In this work, we also use this knapsack formulation based on which we develop our algorithms.

The number of accessed objects can be very large and the available cache space is expected to be much smaller. For example, the on-disk data size can be in the order of TBs or PBs, but the available cache space could be in the order of GBs. The key challenge in such an environment is to choose the "best" subset of objects to cache (hotset) in order to improve overall performance.

Let the objects be denoted by $i = 1, \ldots, n$, denote the size of object i by s_i and let $P_i(t)$ be the probability that the object i will be referenced at time t. Let us denote by c_i, the benefit from the presence in cache (or the cost of a miss) of object i. The benefit c_i may depend on s_i and possibly other characteristics of the object including its source (hard disk, SSD, etc.)

If the cache has a capacity C, then an optimal set $M(t)$ of items to be in cache at time t is one that maximizes the total benefit of having the objects in the cache:

$$\sum_{i \in M(t)} c_i P_i(t)$$

subject to the capacity constraint

$$\sum_{i \in M(t)} s_i \leq C \ .$$

We now define the notion of *weight* of an object, which we later use when describing our caching algorithms.

DEFINITION 2.1. *The weight of an object i at time t is denoted by $W_i(t)$ and is defined as $W_i(t) = c_i P_i(t)$.*

Formally, using the above definition, the exact optimization problem is modeled by the following integer linear programming problem (so-called the knapsack problem), using boolean decision variables x_i, which indicates the presence of object i in the cache:

PROBLEM 2.1.

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{n} [c_i P_i(t)] x_i = \sum_{i=1}^{n} W_i(t), x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} s_i x_i \leq C \\ & x_i \in \{0, 1\} \quad (i = 1, \ldots, n) \ . \end{aligned}$$

As shown in the knapsack formulation above, the objective is to maximize the *total weight* in the cache. This problem is NP-hard [14]. However, an approximate solution can be obtained by relaxing the integrality constraints [14], resulting in the following formulation:

PROBLEM 2.2.

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{n} W_i(t)\, x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} s_i x_i \leq C \\ & 0 \leq x_i \leq 1 \quad (i = 1, \ldots, n) \; . \end{aligned}$$

The relaxation above gives rise to an almost-integral solution as follows. Consider the ratios

$$R_i(t) = \frac{c_i P_i(t)}{s_i} = \frac{W_i(t)}{s_i} \quad (i = 1, \ldots, n) \; . \tag{1}$$

If

$$R_{i_1}(t) \geq R_{i_2}(t) \geq \;\cdots\;,$$

then we pick the largest index J such that

$$\sum_{j=1}^{J} s_{i_j} \leq C$$

and place in cache the set $M(t) = \{i_1, \ldots, i_J\}$.

This solution suggests that in order to determine which objects should be stored in the cache at a future time t, the caching algorithm should maintain the objects in a sorted list according to the ratios $R_i(t), 1 \leq i \leq n$. Then, it should select objects with the highest ratio $R_i(t)$ from the list, and add them in the cache until it is full. This approximate solution, which is based on the order of the ratios $R_i(t)$, is the foundation on which our algorithms are build for making cache insertion, replacement and eviction decisions.

The knapsack formulation presented above requires knowledge of $W_i(t)$, and thus $P_i(t)$, which is the probability that the object i will be referenced at time t. It is obvious that an online algorithm cannot know *a priori* the value of this probability for each object. Our proposed caching algorithms estimate the probability values based on the object accesses observed in the past. As we will show in the following section, different algorithms use different probability estimation formulas.

In this work, we assume that the cost of miss c_i of an object i is proportional to the object's size s_i. This is a reasonable assumption in cases where the object represents one or more files to read from a hard disk or over the network.

3 Caching Algorithms

In this section, we discuss in detail our caching framework for external caches. Our proposed algorithms build upon the knapsack formulation and the approximate solution presented in the previous section. They extend it to take into account the state of the cache over time, and by introducing selective cache insertions to minimize the overhead of inserting objects into the external cache. In our environment, each object can represent an HDFS file or an HDFS directory that consists of multiple files, which are all scanned when the directory is accessed.

3.1 Caching Algorithm Properties

In this section, we present the major characteristics of our caching methods.

- **Online Algorithm**: Our proposed caching algorithms are online algorithms that do not assume any knowledge of the future workload. The caching algorithm is invoked every time an object is accessed. Upon a cache miss, the algorithm decides whether the newly-accessed object should be inserted in the cache, and if there is not enough free space, which cached objects should be evicted in order to accommodate the new object.

- **Estimating the probability of re-access based on the workload history**: As we discussed in the previous section, the set of objects selected to reside in the cache at a *future time t* depends on the probability of accessing each object at time t, namely $P_i(t)$. In practice, online caching algorithms cannot know *a priori* this probability for a future point in time. However, at *current time u*, they can statistically or heuristically estimate the probability based on their knowledge of the workload history up to time u. Let's denote this probability as $p_i(u)$. Our algorithms build on the knapsack approximation presented in the previous section by making the assumption that $P_i(t) \simeq p_i(u)$. Thus, we can also assume that $W_i(t) \simeq w_i(u) = c_i p_i(u)$ and that $R_i(t) \simeq r_i(u) = w_i(u)/s_i$.

  Moreover, our caching algorithms assume that the probability function $p_i(u)$ has the following property:

  ASSUMPTION 3.1. *If $p_i(u) > p_j(u)$ at a time u then $p_i(u+\Delta u) > p_j(u+\Delta u)$ for all objects i, j that have not been accessed during the interval $(u, u+\Delta u]$. Thus, if $r_i(u) > r_j(u)$, then $r_i(u+\Delta u) > r_j(u+\Delta u)$.*

Consider a sorted list that contains information about the objects residing in the cache at time u. The objects in the list are sorted in ascending order of the ratio $r_i(u)$ as discussed in Section 2. Let's assume that we want to maintain the list sorted as objects are accessed over time and their probabilities of re-access change. The next object access happens at time $u + \Delta u$. According to Assumption 3.1, the relative order of those objects in the list that were not accessed during the time interval $(u, u + \Delta u]$, does not need to change. Only the position of the currently-accessed object needs to be updated. In this way, we can avoid re-sorting the whole list after each object access.

- **Selective Cache Insertions**: Typically, caching algorithms such as the `LRU-K` method, are focused on which objects should be evicted from the cache to accommodate a newly-accessed object. These algorithms always insert the newly-accessed object in the cache. However, this policy is not applicable to our setting, where the cache is external, because cache insertions are performed by an external process, which is not part of the query engine. This process competes for resources (e.g., I/O bandwidth) with the query engine and can actually slow down the processing of the workload. In Section 5, we present experimental results that highlight this problem.

  To overcome this problem, our caching algorithms selectively perform insertions using a greedy heuristic called the `weight heuristic`. As shown in the knapsack formulation, the objective function aims at maximizing the *total weight* in the cache. The `weight heuristic` attempts to do that by inserting objects in the cache only when the *total weight* in the cache would not decrease because of the insertion operation. More specifically, upon an object access and a subsequent cache miss, the `weight heuristic` compares the weight of the newly-accessed object with the sum of the weights of the objects that need to be evicted from the cache in order to accommodate the new object. The object is inserted into the cache only if the replacement operation results in an increase of the *total weight* in the cache.

3.2 Caching Algorithm Template

In this section, we provide a template algorithm that is invoked each time an object is accessed. Our proposed `SLRU-K` and `EXD` algorithms specialize this template by providing their own definitions of $p_i(u)$, and thus $w_i(u)$ and $r_i(u)$. The pseudocode of the algorithm is shown in Algorithm 1. We use a global integer counter

Algorithm 1: Caching Algorithm Template

```
Data: Accessed Object $b$, Size of object $b$: $s_b$, Used, Capacity, CacheState, History
Result: true if $b$ is inserted in the cache, false otherwise
Time++;
If $b$ is contained in History then retrieve the latest information about this object, otherwise create a new History entry for $b$;
Set object's $b$ last access time to Time;
if Object $b$ is in the cache then
    // Cache Hit
    Remove $b$ from the CacheState and re-insert it with ratio $r_i$(Time);
    return false;
else
    // Cache Miss
    if $s_b$ + Used $\leq$ Capacity then
        // Object $b$ fits in the cache
        Insert object $b$ in the CacheState with ratio $r_i$(Time) ;
        Used=Used+$s_b$;
        Insert $b$ into the cache;
        return true;
    else
        // Object $b$ does not fit in the cache
        // Evaluate whether $b$ should be inserted in the cache using the weight heuristic
        Compute the object's $b$ weight $w_b$(Time);
        Maintain the sum of the weights of the objects that will be evicted as sumWeights = 0;
        Set freeSpace = Capacity - Used;
        foreach object next in CacheState in ascending order of ratio do
            if sumWeights + $w_{next}$(Time) < $w_b$(Time) then
                sumWeights = sumWeights + $w_{\text{next}}$(Time);
                freeSpace = freeSpace + $s_{\text{next}}$;
                Add next to the Eviction List.;
                if freeSpace $\geq s_b$ then
                    exit the loop;
        if freeSpace $< s_b$ then
            // Haven't found candidates for eviction
            Object $b$ is not inserted in the cache;
            return false;
        else
            // Found candidate objects for eviction
            Evict from the cache all the objects in Eviction List;
            Insert object $b$ in the CacheState with ratio $r_i$(Time) ;
            Insert $b$ into the cache;
            return true;
```

Time to simulate time which is incremented each time an object is accessed.

The algorithm maintains two data structures: the CacheState and the History. The CacheState contains all the information about the objects that are currently in the cache, including the ratio $r_i(u)$ at time u and their size. The CacheState is implemented as a list sorted by $r_i(u)$ in ascending order. In practice, by making use of a probability function that satisfies Assumption 3.1, a caching algorithm can maintain the correct sorted order as objects are accessed, without updating the ratios of all the objects in the cache each time.

The History contains metadata about all the objects that have been accessed in the past, such as their size, and time of last access, and can be implemented as a hash table keyed by the objects. Since the History grows over time, one can restrict the number of entries in this data structure, or remove from History objects that have not been accessed for a long period of time.

Let us consider a cache of size `Capacity`. Let `Used` be the current size of the cache used to store objects. When an object *b* is accessed, the `Time` counter is incremented by 1, and if the object is contained in `History` then the latest metadata about the object is retrieved. If the object *b* is not present in `History` then a new entry is created for it (Lines 1-3).

The algorithm then checks whether the object is already in the cache (*cache hit*) or not (*cache miss*). In case the object *b* is already in the cache, the algorithm needs to update the object's corresponding metadata, namely, its latest access time as well as its ratio $r_b(\texttt{Time})$. Note that since the `CacheState` is implemented as a list sorted by the ratios of the cached objects, we need to remove object *b* from the list, update its ratio, and then re-insert it to keep the correct sorted order (Lines 4-6). We would like to emphasize that if the probability function of the algorithm satisfies Assumption 3.1, then we do not need to update the ratios of the cached non-accessed objects to reflect the new value of the `Time` counter since the sort order is correctly maintained.

If the object is not contained in the cache (*cache miss)*, then the algorithm checks whether there is enough free space in the cache to accommodate the object. If so, the object is inserted into the cache (Lines 8-12). Otherwise, the algorithm uses the `weight heuristic` to identify whether the newly-accessed object should be cached.

As we previously discussed, the `weight heuristic` attempts to minimize insertions in the cache, since they can negatively affect the workload performance. The heuristic applies a greedy approach to maximize the *total weight* of the objects in the cache each time a cache insertion decision needs to be made. Following the approximate solution presented in Section 2, the heuristic traverses the objects stored in `CacheState` in *ascending* order of ratios, attempting to identify potential candidates for eviction in order to accommodate object *b*. The heuristic maintains a list of potential candidate objects for eviction, namely `Eviction List`. At every step, the algorithm checks whether by adding the object currently under consideration to the `Eviction List`, the total weight of the candidate objects for eviction would be less than the weight of the newly-accessed object *b*. In this case, the object currently under consideration is added to the candidate `Eviction List` (Lines 18-23). Otherwise, the object currently under consideration is not added to the `Eviction List`, and the algorithm proceeds with the next object in the sorted list. The heuristic terminates if enough space for the newly-accessed object is found (Lines 22-23), or if all the objects in the list have been examined. If the total size of the objects in the `Eviction List` is enough, then object *b* is inserted in the cache (Lines 24-31).

3.3 Estimating the Probability of Access

In this section, we present in detail the `SLRU-K` and `EXD` algorithms for external caches. Both algorithms are instantiations of the template presented in the previous section but utilize different definitions of $p_i(u)$. Because of the different nature of the probability functions, the two algorithms maintain different types of metadata per object. More specifically, the `EXD` algorithm requires fewer metadata items per object than the `SLRU-K` algorithm.

3.3.1 The SLRU-K algorithm

The `Selective LRU-K (SLRU-K)` algorithm is an extension of the LRU-K algorithm that takes into account the variable size of the objects. As opposed to `LRU-K`, the `SLRU-K` algorithm does not insert each accessed object into the cache, but rather selectively places objects in the cache using the `weight heuristic`.

For each object i, the `SLRU-K` algorithm maintains a list of times of its K most recent accesses sorted in descending order, namely $L_i = [u_{i1}, ..., u_{iK}]$ where each u_{ij} is equal to the value of the `Time` counter at the jth most recent access of the object i. Thus, the time of the last access of the object is represented by u_{i1} and the time of the Kth most recent access is represented by u_{iK}. This list is updated when the object is accessed, by introducing a new value (time of last access) in the head of the list and dropping the last value, if needed, in order to keep the list limited to at most K values.

DEFINITION 3.1. *For a given object i and current time u, let $T_i(u) = u - u_{iK} + 1$ be the number of object accesses since object i's Kth most recent access.*

The estimate used by the `SLRU-K` algorithm for $p_i(u)$ is based on a model as follows. Suppose $X_u, X_{u-1}, X_{u-2}, \dots$ are independent and identically distributed Bernoulli random variables, each with success probability p. Let T be the random variable whose value is determined from $\sum_{i=0}^{T-1} X_{u-i} = k$ and $X_{u-T+1} = 1$. Thus, T is the cardinality of the smallest interval of consecutive random variables $X_u, X_{u-1}, \dots, X_{u-T+1}$ that contains the first K successes in the above sequence. We wish to estimate p by observing the value of T. It follows that the maximum-likelihood estimate of p is $p = \frac{K}{T}$. Based on this model, the `SLRU-K` algorithm estimates the probability that object i will be accessed at time $u+1$ as

$$p_i(u) = \frac{K}{T_i(u)} , \tag{2}$$

where $T_i(u)$ is the total number of accesses in the interval (see above) that includes the K most recent accesses of object i until time u.

Note that the estimate $p_i(u)$ is changing over time as more accesses are happening, and the value of $T_i(u)$ changes. The `SLRU-K` algorithm takes into account the new values of these estimates since the list of the last K accesses of each object is updated.

PROPOSITION 3.1. *The probability function of the* `SLRU-K` *method has the property described in Assumption 3.1.*

Proof.

$$\begin{aligned}
& p_i(u) > p_j(u) \\
\Leftrightarrow\quad & K/T_i(u) > K/T_j(u) \\
\Leftrightarrow\quad & T_i(u) < T_j(u) \\
\Leftrightarrow\quad & u - u_{iK} + 1 < u - u_{jK} + 1 \\
\Leftrightarrow\quad & u + \Delta u - u_{iK} + 1 < u + \Delta u - u_{jK} + 1 \\
\Leftrightarrow\quad & T_i(u+\Delta u) < T_j(u+\Delta u) \\
\Leftrightarrow\quad & k/T_i(u+\Delta u) > k/T_j(u+\Delta u) \\
\Leftrightarrow\quad & p_i(u+\Delta u) > p_j(u+\Delta u)\ .
\end{aligned}$$

□

3.3.2 The EXD algorithm

In this section, we present the novel algorithm we developed for external caches, namely, the `Exponential-Decay (EXD)` caching algorithm. The algorithm implements the template presented in Section 2, and makes use of a single parameter (a) that determines the weight of frequency vs. recency of data accesses. In this section, we focus on how the `EXD` algorithm approximates the probability $p_i(u)$.

The algorithm maintains a `score` for each object. The `score` $S_i(u)$ of object i at current time u is defined as follows.

DEFINITION 3.2. Denote by

$$u_{i1} > u_{i2} > \cdots$$

the time points at which object i was previously accessed, then

$$S_i(u) = e^{-a(u-u_{i1})} + e^{-a(u-u_{i2})} + \cdots\ ,$$

where $a > 0$ is a constant whose value is yet to be determined.

As shown, the `score` of an object depends on the value of the parameter a. The value of this parameter essentially determines how recency and frequency are combined into a single score. The larger the value of a, the more emphasis on recency versus frequency. The value of a can also be chosen adaptively as we will describe in Section 3.4. The `EXD` algorithm makes the following assumption:

ASSUMPTION 3.2. *For a given object i, at the current time u, $S_i(u)$ is proportional to $p_i(u)$.*

Notice that our proposed caching algorithm does not require exact knowledge of the values of $p_i(u)$ of the accessed objects. It rather needs to know the relative order of the ratios $r_i(u)$ of all different objects. For this reason, the `EXD` algorithm substitutes the object's probability function $p_i(u)$ with the object's `score` $S_i(u)$ in Algorithm 1.

It follows that at any given point in time u, the `EXD` algorithm needs to compute the `score` $S_i(u)$ of the objects. The following proposition describes how we can efficiently compute the `score` of an object at a specific point in time, given only the time of its last access, and the corresponding `score` at that time. Note that, unlike the `SLRU-K` algorithm which needs to maintain the last K access times for each object, the `EXD` algorithm reduces the memory footprint by keeping only the time of the last access of each object.

DEFINITION 3.3. *For an object i, the score $S_i(u_{i1} + \Delta u)$ can be calculated if we only keep the most recent time of access u_{i1} and the score $S_i(u_{i1})$.*

Proof. Obviously, if object i is not accessed during the interval $(u_{i1}, u_{i1} + \Delta u]$, then

$$S_i(u_{i1} + \Delta u) = S_i(u_{i1}) \cdot e^{-a\Delta u} \tag{3}$$

and if it is accessed at time $u_{i1} + \Delta u$ for the first time after time u_{i1}, then

$$S_i(u_{i1} + \Delta u) = S_i(u_{i1}) \cdot e^{-a\Delta u} + 1 \ . \tag{4}$$

□

It follows that the score $S_i(u)$ can be calculated for any time $u > u_{i1}$ before the next object access. Furthermore, the scores decay exponentially and can be approximated by zero after they drop below a certain threshold. This allows us to stop maintaining history for objects that have not been accessed for a long time.

PROPOSITION 3.2. *The scoring function (and thus the probability function) of the* `EXD` *method has the property described in Assumption 3.1.*

Proof. Equation 3 implies that between object accesses, the order on the set of objects (that have not been accessed during the time interval) implied by the ratios does not change, i.e.,

$$\begin{aligned} & S_i(u) > S_j(u) \\ \Leftrightarrow \quad & S_i(u) \cdot e^{-a\Delta u} > S_j(u) \cdot e^{-a\Delta u} \\ \Leftrightarrow \quad & S_i(u+\Delta u) > S_j(u+\Delta u) . \end{aligned}$$

□

Algorithm 2: Adaptor

```
Data: boolean CacheHit, boolean ObjectInserted, long objectSize
Result: new value of algorithmic parameter newParameter
1  eventNo++;
2  Update the BHR(currentParameter) and BIR(currentParameter) based on the values of CacheHit,
   ObjectInserted, and objectSize;
3  if (eventNo == maxEventsPerRound) then
       // end of current round
4      eventNo = 0;
       // Update the BHR and BIR values taking into account all the rounds so far
5      BHR(currentParameter) = weightedAverage(previousBHR(currentParameter), BHR(currentParameter));
6      BIR(currentParameter) = weightedAverage(previousBIR(currentParameter)), BIR(currentParameter));
       // Select the new value of the parameter
7      Group the parameters in CandidateValues according to their corresponding BHR observed so far;
8      if (no time for exploration) then
9          selectedGroup = pick group with highest representative BHR;
10     else
11         selectedGroup = pick group with probability proportional to its BHR;
12     newParameter = pick the parameter value in selectedGroup with the minimum BIR value;
13     return newParameter to the caching algorithm;
14 else
       // not the end of current round
15     newParameter = current value of the parameter;
16     return newParameter to the caching algorithm;
```

3.4 Adaptive SLRU-K and EXD

Both the EXD and the SLRU-K algorithms depend on parameters (*a*, *K*). The behavior of the algorithms can significantly change based on the values of *a* and *K*. As we will show in Section 5, there is no single value of *a* (or *K*) that works well across all possible workloads.

Figuring out the best value of the algorithmic parameter is difficult for two reasons: (1) The optimal value of the parameter depends heavily on the workload access pattern, and (2) The workload access pattern is not stable over time. In this section, we present an adaptive algorithm (`Adaptor`) that automatically adjusts the value of the algorithmic parameter in order to improve overall performance.

The `Adaptor` can be used with both the `SLRU-K` and the `EXD` methods. It operates along with the caching algorithm and exchanges information with it. Each object access is treated as an `event`. At every `event`, the caching algorithm informs the `Adaptor` whether the `event` was a cache miss or a cache hit, and whether the object was inserted into the cache. The `Adaptor` uses this information to adjust the algorithmic parameters over time.

The `Adaptor` takes into account two metrics when making decisions about the value of the algorithmic parameter. The *primary metric* is the *byte hit ratio (BHR)* which is a standard comparative performance metric used in prior work on caching variable-size objects [6, 29, 2, 27]. The *BHR* is the fraction of the requested bytes that was served from the cache. The higher the *BHR*, the fewer I/O requests need to be made, and the greater the overall performance. As in previous work, our primary goal is to maximize the `BHR`.

In an external caching system, such as in HDFS cache, cache insertions compete for resources with the process that needs to access the data, and thereby slow down the workload. To quantify the overhead of each algorithm with respect to cache insertions, we introduce a *secondary metric*, namely the *byte insertion ratio (BIR)*. The *BIR* is the fraction of the requested bytes that the caching algorithm decided to insert into the cache.

In our environment, it is desirable to maximize the *BHR* so that the hot set is always cached while maintaining a low *BIR* if possible. Our `Adaptor` constantly evaluates the behavior of the caching algorithm by measuring these metrics, and its *primary goal* is to maximize the *BHR*. From all the values of the algorithmic parameter that maximize the *BHR*, the `Adaptor` prefers the one that minimizes the *BIR*, since it reduces the cost of insertions in the cache.

The pseudocode for the `Adaptor` is presented in Algorithm 2. The algorithm uses a set of pre-defined parameter values, namely *CandidateValues*. In case of the `SLRU-K` algorithm, the *CandidateValues* set contains the following values for the K parameter: $1, 2, 4, 6, 8$. In the case of the `EXD` algorithm, the *CandidateValues* set contains six a values equally-spaced in the log space with $a_{min} = 10^{-12}$ and $a_{max} = 0.3$. These values cover a large range of potential parameter instantiations that can successfully be applied in many workload scenarios. For each potential value of the algorithmic parameter $i \in CandidateValues$, the `Adaptor` maintains the observed $BHR(i)$ and $BIR(i)$ achieved with the value i so far.

The algorithm operates on `rounds` that consist of a fixed number of `events`.

After every `event`, the `Adaptor` updates the *BHR* and *BIR* values observed for the current value of the parameter (*currentParameter*), based on the information received from the caching algorithm (Line 2).

When the last `event` of the `round` is processed, the *BHR* and the *BIR* values that correspond to the current parameter value are updated using a weighted average over the observed *BHR* and *BIR* values across all `rounds`, giving more emphasis on the observations of the last `round` (Lines 3-6). The `Adaptor` then re-evaluates the value of the algorithmic parameter. The re-evaluation process consists of three steps. In the first step, the `Adaptor` groups the parameter values of the *CandidateValues* set, according to their observed *BHR* so far. Parameter values with *BHR* values within a certain threshold of each other are placed in the same group (Line 7). Each group has a representative *BHR* value, which is the average of the *BHR* of its members. In the next step, the `Adaptor` picks the group with the highest representative *BHR* (Lines 8,9). Occasionally, at this step, the `Adaptor` selects a group with probability proportional to the *BHR* of the group (Lines 10,11). This happens so that the parameter space is explored by observing the behavior of the caching algorithm for different values of the parameter. After a group has been selected, the `Adaptor` selects a member of this group by taking into account the *BIR* values that have been achieved so far by the members of the group. More specifically, it picks the parameter value that has resulted in the lowest *BIR* so far (Line 12).

After the value of the parameter has been selected, the `Adaptor` informs the caching algorithm of the new value (Lines 13,16). The caching algorithm, then, updates the ratios of the objects in the `History` and the `CacheState` to reflect the new value.

4 System Implementation

In this section, we briefly describe the implementation of our caching algorithms in Big SQL [5], IBM's SQL-on-Hadoop offering, which is part of the IBM$^{®}$ InfoSphere$^{®}$ BigInsights^{TM} data platform. We use the HDFS cache mechanism that is part of the HDFS file system, and is supported by the platform.

Big SQL [5] leverages IBM's state-of-the-art relational database technology to execute SQL queries over HDFS data, supporting all the common Hadoop file formats; text, sequence, Parquet and ORC files. The Big SQL coordinator compiles and optimizes the query. The database workers read HDFS data directly and execute relational operations.

A fundamental component in Big SQL is the scheduler service, which assigns HDFS blocks to database workers for processing on a query by query basis. The scheduler identifies where the HDFS blocks are, and decides which database workers to include in the query plan. The assignment is done dynamically at run-time to accommodate failures: scheduler uses the workers that are currently up and running. In case of partitioned tables, which are common in SQL-on-Hadoop environments, selection predicates are pushed down to the scheduler to eliminate partitions that are not relevant for a given query. As the scheduler is aware of which data objects are accessed for each query, we incorporated our caching algorithms in the scheduler service.

The caching algorithm operates at the level of table partitions, considering unpartitioned tables as consisting of a single partition. While each partition may itself consist of multiple HDFS files of different sizes, the caching algorithm maintains metadata (see Section 3) per-partition rather than per-file to minimize memory footprint. For every scan operation in a query, the Big SQL scheduler first eliminates unnecessary partitions, and then invokes the caching algorithm to decide which partitions to insert into the HDFS cache. The scheduler uses the appropriate HDFS APIs [13] to instruct HDFS to cache a partition. Note that HDFS performs the actual cache insertions, not Big SQL. The Big SQL scheduler runs a separate thread for the `Adaptor`, in case of adaptive algorithms. This thread communicates with the thread running the caching algorithm to exchange the necessary information (see Section 3.4).

The HDFS cache [13] implements its own algorithms to decide which replica of a given block will be cached, and in which DataNode. During query execution, the Big SQL scheduler always attempts to assign data to worker nodes optimizing for data locality in a best effort fashion, giving priority to memory locality, and then disk locality. More specifically the scheduler, gathers the locations of all the replicas of a given block that will be accessed by the query, and attempts to first assign the in-memory replicas to the workers that host them, then assigns the local on-disk replicas, and finally incorporates accesses to remote replicas.

Data on HDFS may occasionally change. For example, deletion of files, file appends, or file additions in a table or partition can be performed without going through the Big SQL interface. For this reason, the caching algorithm maintains a timestamp for each partition (table) in the cache. The timestamp is the time of the latest modification of all the files that comprise the partition (or table). When the partition (or table) is accessed again, the algorithm checks the latest modification time for this data to identify potential data changes since the last time this data was accessed. In case there has been a change, the algorithm compares the new size of the data with the size of the previous access. If the new size is smaller than the one stored in the metadata, then one or more deletion operations have been performed and some files would no longer reside in the HDFS cache. This is because, when a cached HDFS file is deleted, HDFS automatically removes it from the HDFS cache. In this case, the caching algorithm only updates its metadata (latest modification time, new size of the data). In case of file appends or additions, the partition (or table) is removed from the cache, and we try to re-insert it into the cache with its new data size.

5 Experimental Evaluation

In this section, we provide a comprehensive evaluation of our proposed algorithms with state-of-the-art caching policies. We begin with a simulation study which demonstrates how well our algorithms adapt to various data access patterns. Simulation studies, while admittedly artificial, give us fine-grained control on the workload pattern, and allow us to reason about the comparative performance of the algorithms without being clouded by incidental system implementation or hardware details (e.g., CPU efficiency, I/O and network bandwidth). For this reason, simulation studies have been extensively used to evaluate caching algorithms in prior work [23, 25, 18, 6, 20, 31, 17, 27]. We, then, back up these findings with experimental results running IBM's Big SQL on various workloads in a cluster environment in Section 5.2.

5.1 Simulation Study

5.1.1 Simulator Design

To evaluate our algorithms under various scenarios, we implemented a simulation framework that generates synthetic workloads accessing data objects whose sizes are drawn from various distributions. The simulator framework is similar to the one used in [10] in the context of Map-Reduce, but is extended to accommodate objects of different sizes.

The simulator framework consists of a data object generator and three workload generators. The data generator creates a database D, consisting of objects with different sizes, drawn from various distributions such as fixed (constant), uniform, log-normal, and log-uniform distributions. The workload generators generate sequences of accesses to these objects.

The frequency-based workload generator W_f emphasizes the *frequency* of object accesses. For example, in a partitioned table in a SQL-on-Hadoop system, it is likely that the last year's partitions will be more frequently accessed than partitions from a decade ago. W_f generates a workload in which objects from a predefined subset, namely, the *coreSet*, are accessed more frequently than the remaining objects. The W_f generator either picks an object at random, uniformly, from the *coreSet* (with probability p_f), or uniformly from D (with probability $1 - p_f$).

The recency-based workload generator W_r emphasizes the *recency* of object accesses. More specifically, W_r keeps a sliding window, called the *stickiness window*, over the sequence of object accesses. The stickiness window size determines the number of recently-accessed objects that are more likely to be accessed again in future requests; the larger the stickiness window size, the longer an object will persist in the workload. The W_r generator either picks an object at random, uniformly, from the *stickiness window* (with probability p_r), or uniformly from D. In the latter case, the selected object will be inserted into the *stickiness window*, applying an LRU eviction policy if necessary.

The hybrid workload generator W_h generates a workload with both *frequency* and *recency* characteristics. Given a probability parameter p_h, to generate each access, W_h invokes either W_r with probability p_h or W_f with probability 1-p_h. In this case, the hotset evolves over time based on the recent accesses, but it also contains some objects, which are accessed more frequently than others.

5.1.2 Evaluation

In this section, we present our results using our simulation framework. We now present our experimental setting:

- **Data**: We generated a database of size 1M, using objects whose sizes were drawn uniformly from the range $[1, 1999]$.[1] This distribution allows us to evaluate scenarios with a wide range of data object sizes. In the interest of brevity, we omit similar results that we obtained by varying the distribution width, and choosing other size distributions such as log-normal or log-uniform.
- **Evaluation Metrics**: We evaluate various caching algorithms using the *byte hit ratio (BHR)* and the *byte insertion ratio (BIR)* metrics (see Section 3.4). When comparing algorithms, our primary goal is to maximize the *BHR*. Among all the algorithms that maximize the *BHR*, we prefer the one that minimizes the *BIR*,

[1]The unit of data size does not affect the results.

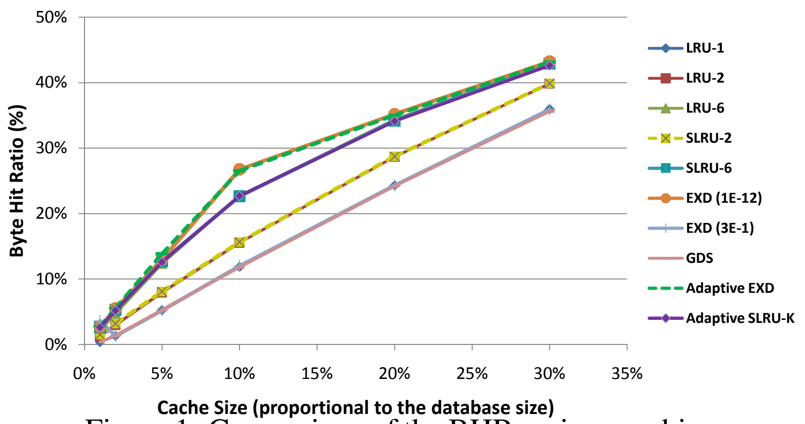

Figure 1: Comparison of the BHR various caching algorithms using the W_f generator

since it minimizes the cost of cache insertions.

- **Caching Algorithms**: We evaluate our proposed `EXD` and `SLRU-K` algorithms for various values of the parameters a and K, as well as their adaptive versions, namely, `Adaptive EXD` and `Adaptive SLRU-K`. The well-known `LRU-K` method extended to accommodate variable-size objects has been evaluated in the context of web caching [6] only when $K = 1$. We further evaluate the extended `LRU-K` algorithm for multiple values of K. We note that, the main difference between the `LRU-K` and the `SLRU-K` algorithms is that the former inserts every accessed data object into the cache whereas the latter performs selective cache insertions. Finally, we evaluate the `GreedyDual-Size (GDS)` algorithm [6]. The `GDS` algorithm is developed for web caching, is parameter-free, is able to accommodate various file sizes and has been shown to outperform various algorithms for web caches [6]. To the best of our knowledge, the `GDS` algorithm is the state-of-the-art algorithm for variable file sizes in the context of web caching.

Frequency-based workload In our first experiment, we use the W_f workload generator. The *coreSet* contains 10% of the data objects in D. The probability p_f is set to 0.2. We also performed experiments setting the *coreSet* size to $1\%, 2\%, 5\%, 20\%$ of the data objects and p_f to 0.1, 0.3, and 0.5. These values produced similar results, hence are omitted.

Figures 1 and 2 present the *byte hit ratio* and *byte insertion ratio* for different caching algorithms, and cache sizes. Regarding the *BHR* metric, we observe the following:

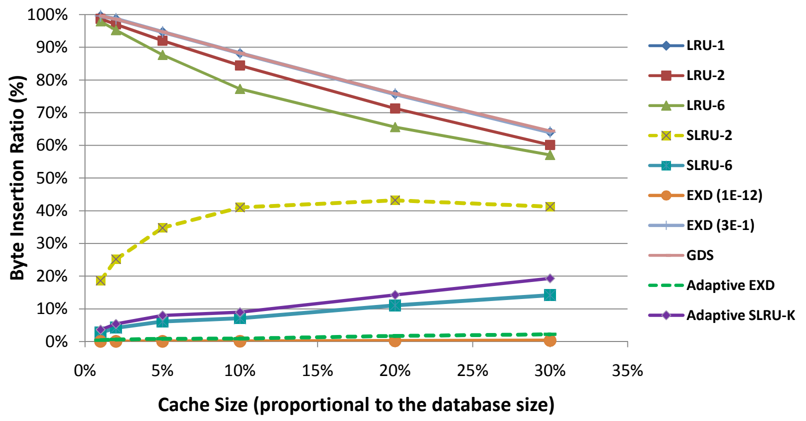

Figure 2: Comparison of the BIR various caching algorithms using the W_f generator

1. The value of K significantly affects the behavior of both the `LRU-K` and the `SLRU-K` with greater values of K providing higher *BHR*. This is because the lower the value of K, the more emphasis is given by the algorithm on the recency than the frequency of data accesses, which is the focus of the W_f generator.
2. The `LRU-K` and the `SLRU-K` algorithms have very similar *BHR* for the same value of K, since both algorithms use the same probability function, and thus identify similar hotsets. However, the performance of the two algorithms differs with respect to the *BIR* metric as we discuss next.
3. The performance of the `EXD` algorithm with respect to the *BHR* metric varies significantly as the parameter a varies. More specifically, the lower the value of a the better the *BHR* since a lower a gives more emphasis on the frequency of object accesses.
4. The `GDS` algorithm has similar *BHR* values to the `LRU-1` and `EXD`(0.3) algorithms, both of which give emphasis to recency.
5. Finally, the parameter-free `Adaptive EXD` and the `Adaptive SLRU-K` algorithms are able to identify the correct values of K and a that result in high *BHR* for all cache sizes. Notice that the `Adaptive EXD` provides slightly better *BHR* values.

Regarding the *BIR* metric, we can observe the following:

1. The `SLRU-K` algorithm has significantly lower *BIR* than the `LRU-K` algorithm for the same value of K due to the `weight heuristic` that attempts to avoid unnecessary insertions whereas `LRU-K` performs an object insertion for every cache miss.

2. As the value of K decreases, the *BIR* of the `SLRU-K` metric decreases. When the value of K is small, the algorithm gives more emphasis on the recency of data accesses, trying to maintain the most recently accessed data in the cache, thus incurring a large number of cache insertions. Obviously, this behavior is not desirable for workloads generated by the W_f generator.
3. The performance of the `EXD` algorithm with respect to the *BIR* metric varies with the parameter a. More specifically, the lower the value of a, e.g., 10^{-12}, the better the *BIR* since a lower a gives more emphasis on the frequency of object accesses, and less on the recency of accesses.
4. The `GDS` algorithm has similar *BIR* values to the `LRU-1` and `EXD`(0.3) algorithms, both of which give emphasis to recency.
5. The adaptive `Adaptive EXD` and the `Adaptive SLRU-K` algorithms are able to identify the correct values of K and a that result in low *BIR* for all cache sizes. Notice that the `Adaptive EXD` can provide better *BIR* values than the `Adaptive SLRU-K`.
6. Another observation is that the trends for the *BIR* metric are not the same across different algorithms. Some algorithms, like `LRU-K` have a very high *BIR* at small cache sizes, which decreases as the cache becomes larger. This is because the larger the cache size, the more objects fit in the cache and thus cache insertions are avoided. However, other algorithms such as `EXD` and `SLRU-K` have very low *BIR* for small cache sizes. The reason is that the `weight heuristic` that these algorithms use, identifies that it is not worth to keep replacing objects in the cache when the hotset does not fit in the cache, but it is more beneficial to maintain a few popular objects in the cache. This policy results in slightly higher *BHR* than `LRU-K` for small cache sizes, and much lower *BIR*.

Overall, we observe that the `Adaptive EXD` and the `Adaptive SLRU-K` algorithms learn the correct values of the parameter, and thus produce high *BHR*, and low *BIR* for all cache sizes.

Recency-based workload In the next experiment, we use the W_r workload generator and set p_r to 0.2 . The *stickiness window* contains 10% of the data objects in D. Other values produced similar results, and hence are omitted.

Figures 3 and 4 present the *BHR* and the *BIR* values for various caching algorithms and cache sizes. In this case, the algorithms that value recency more than frequency in their caching decisions (such as `LRU-1`, `EXD`(0.3), `GDS`) exhibit the best *BHR* performance. Notice that these algorithms also have the highest *BIR* values. This is not surprising since insertions are necessary in order to keep up

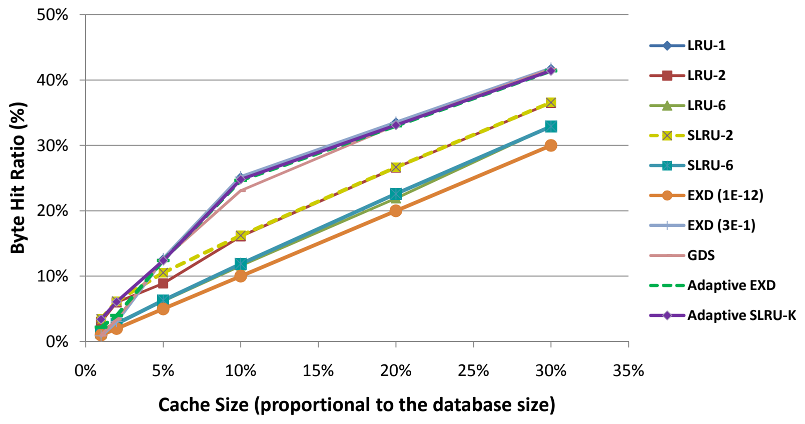

Figure 3: Comparison of the BHR various caching algorithms using the W_r generator

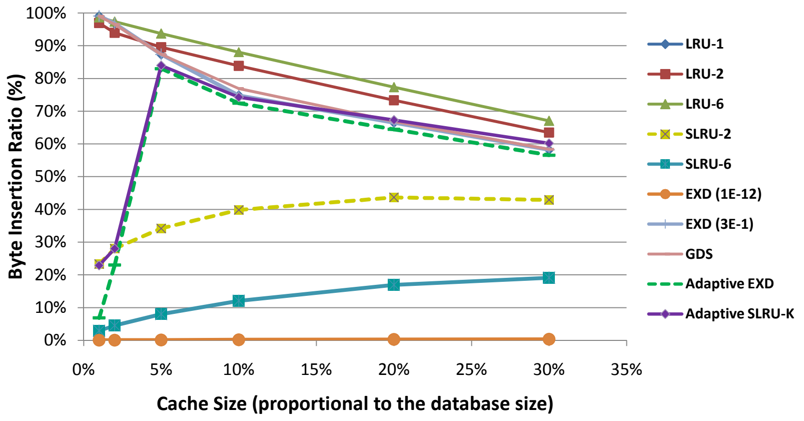

Figure 4: Comparison of the BIR various caching algorithms using the W_r generator

with hotsets that are based on the recency of data accesses. In other words, in order to observe high *BHR* values on workloads with hotsets that mainly include the most recent data accesses, an algorithm must not avoid insertions; otherwise, accesses on the hot data will result in cache misses. Note however, that a high *BIR* value does not necessarily mean that the "correct" data is actually cached. For example, the `LRU-2, LRU-6` algorithms have lower *BHR* than `LRU-1`, although they have similar or higher *BIR* value. This is because they give less emphasis on the recency of data accesses.

Another observation is that the `Adaptive SLRU-K` and the `Adaptive EXD` algorithms are able to achieve very good *BHR* values. The *BIR* values of these algorithms are very low for small cache sizes, and then significantly increase for

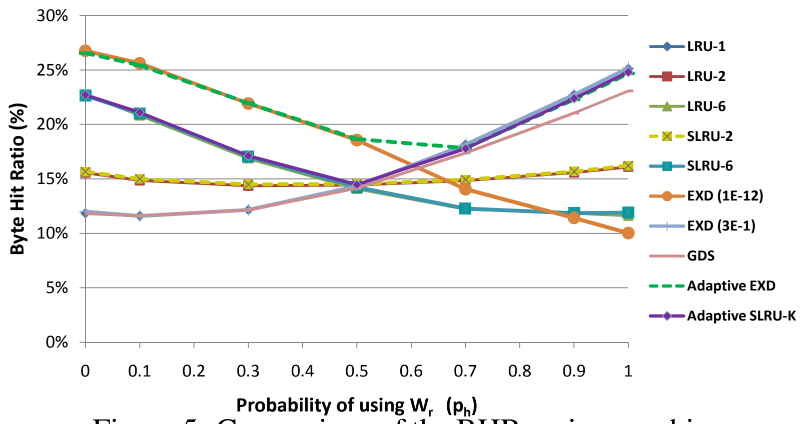

Figure 5: Comparison of the BHR various caching algorithms using the W_h generator

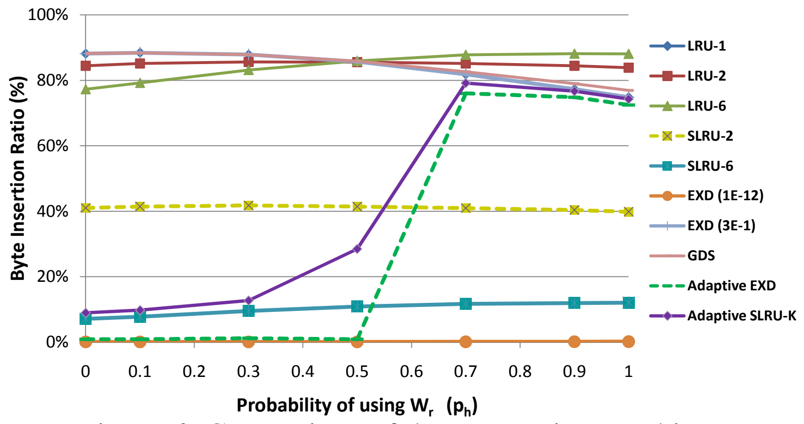

Figure 6: Comparison of the BIR various caching algorithms using the W_h generator

cache sizes greater than 5% of the database size. The reason for this behavior is that when the hotset is much larger than the available cache space, the adaptive algorithms make the decision to cache a small set of objects and do not replace them frequently. In this way, they are able to get a higher *BHR* than algorithms like `EXD(0.3)`, which generally behaves very well for recency-based workloads. As the cache size increases, and more hot data can be accommodated in the cache, the adaptive algorithms do not avoid insertions in order to always cache the evolving hotset. In summary, the adaptive `Adaptive EXD` and the `Adaptive SLRU-K` algorithms produce high *BHR* for all cache sizes by making correct cache insertion decisions.

Hybrid workload In our final experiment, we use the W_h workload generator. In this example, the *coreSet* as well as the *stickiness window* contain 10% of the data objects in D, and their corresponding probabilities p_f and p_r are both set to 0.2. Initially, the *coreSet* and the *stickiness window* contain different objects. We set the cache size to 10% of the database size (other cache sizes produced similar results), and vary the probability of using the W_r generator (p_h).

Figures 5 and 6 show the *BHR* and the *BIR* values for various values of p_h. When $p_h = 0$, only the W_f generator is invoked and thus the algorithms that value frequency such as `LRU-6`, `SLRU-6` and `EXD`(10^{-12}) produce the best *BHR* values. As the value of p_h increases, and thus the W_r generator also gets invoked, the performance of these algorithms with respect to the *BHR* becomes worse and algorithms such as `GDS`, `EXD`(0.3) start becoming better. When $p_h = 1$, the frequency-based algorithms have the worst behavior, whereas the recency-based algorithms produce the best *BHR*. An interesting point is that the `Adaptive EXD` and `Adaptive SLRU-K` methods are able to adjust the values of K and a so that they can produce good *BHR* results irrespective of the value of p_h. None of the other algorithms exhibit this adaptive behavior, as they are only optimized for special workload characteristics. Similarly, the adaptive methods adjust the number of insertions they perform in order to get the best result. When p_h is high, the adaptive methods perform insertions in order to keep up with the recency characteristics of the hotset. Finally, we again observe that the adaptive `SLRU-K` and `EXD` algorithms have lower *BIR* values than the non-adaptive algorithms when that does not negatively impact the *BHR* values.

5.1.3 Summary

Our simulations use a variety of workloads, varying the extent to which the access pattern of objects is affected by frequency and recency of past accesses to them. Overall, we observed that the basic `SLRU-K` and `EXD` algorithms for different parameters achieve high *BHR* and low *BIR* for different workloads, but none of them individually performs well on all of them. However, the adaptive algorithms, especially the `Adaptive EXD` algorithm, achieve the best balance between *BHR* and *BIR*, effectively producing the lowest *BIR* without negatively affecting the *BHR*. Finally, none of the traditional algorithms can consistently outperform our `Adaptive EXD` algorithm across different workload patterns.

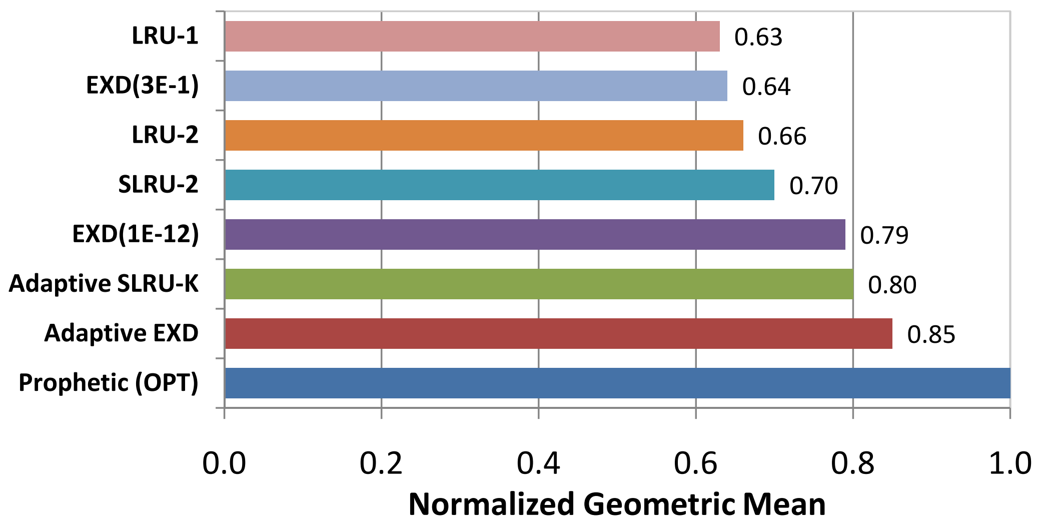

Figure 7: Comparison of various caching algorithms using the TPC-DS like workload

5.2 Cluster Experiments

In this section, we present experiments in a cluster environment using IBM Big SQL and three different workloads.

5.2.1 Experimental Setting

For our experiments, we use a cluster of 10 nodes. One of the nodes hosts the HDFS NameNode, the Big SQL coordinator, the scheduler, and the Hive Metastore. The remaining 9 nodes are designated as "compute" nodes. Every node in the cluster has 2x Intel Xeon CPUs @ 2.20GHz, with 6x physical cores each (12 physical cores total), 8x SATA disks (2TB, 7k RPM), 1x 10 Gigabit Ethernet card, and 96GB of RAM. Out of the eight disks, seven are used for storing HDFS data. Each node runs 64-bit Red Hat Enterprise Linux Server 6.5. We use the implementation of the caching framework described in Section 4, using InfoSphere BigInsights 3.0.1 enterprise release, and test end-to-end system performance.

5.2.2 TPC-DS Like Workload

In this section, we present cluster experiments using a workload inspired by the TPC-DS benchmark[2]. This workload is published by Impala developers[3], and has previously been used to compare the performance of various SQL-on-Hadoop systems (e.g., [28], [11]). The workload consists of 20 queries that include multi-way joins, aggregations, and nested sub-queries. The fact table is partitioned, whereas the small dimension tables are not partitioned. We use a 3TB TPC-DS database, and a 300GB HDFS cache.

We compare the different caching algorithms with a theoretically optimal reference algorithm, which we call the `Prophetic prefetcher`. Before running each query, this algorithm uses prior knowledge of the entire workload trace to prefetch as much of the data accessed by the next query as fits in the cache. As a result, all but 2 of the 20 queries ran entirely in memory. Further, the evaluation of `Prophetic prefetcher` only measures the execution time of the queries, ignoring the time to prefetch the data into memory[4]. For each algorithm, we performed the experiment 3 times using a warm HDFS cache, and report the average over the 3 runs.

Figure 7 shows the geometric mean of the running times of various caching algorithms normalized to the running time of the offline `Prophetic Prefetcher`. As shown, the adaptive algorithms achieve the best performance. The `Prophetic prefetcher` was only about 15% faster than the `Adaptive EXD` algorithm even though it had *a priori* knowledge of the entire workload. The remaining algorithms were not as efficient as the adaptive algorithms. For example, the `LRU-1` algorithm achieved 63% of the `Prophetic Prefetcher's` performance.

The workload's total elapsed time was 2713 seconds when using the `LRU-1` method and 2556 seconds with the `LRU-2` method. The total elapsed time using the `Adaptive EXD` algorithm was 1711 seconds. This is an important difference, especially if we consider that the best possible performance that can be achieved by an offline algorithm is 1544 seconds (`Prophetic Prefetcher`). Figure 8 shows the runtime of each query normalized to the query runtime using the `Prophetic Prefetcher`. Ideally, a caching algorithm should produce query runtimes close to the ones produced by the `Prophetic Prefetcher`. As shown in the figure, the adaptive algorithms generally resulted in query runtimes close to those observed

[2]http://www.tpc.org/tpcds/

[3]https://github.com/cloudera/impala-tpcds-kit

[4]Recall that reading the data into the cache incurs additional cost that needs to be paid by external caches

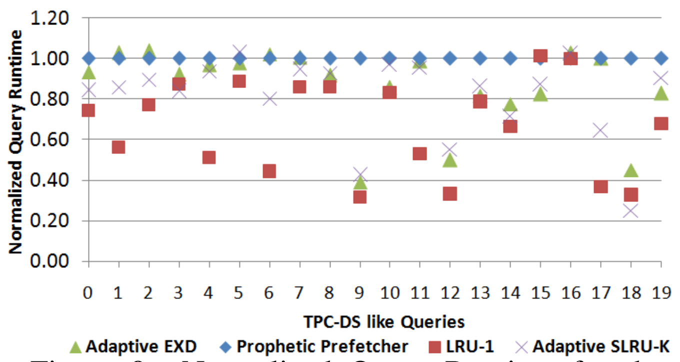

Figure 8: Normalized Query Runtime for the TPC-DS like workload

when the `Prophetic Prefetcher` was used. The `LRU-1` algorithm, on the other hand, did not perform as well as the adaptive methods. When comparing the best performing online algorithm (`Adaptive EXD`) with the `LRU-1` algorithm, we observe that all but one of the queries experienced speedups ranging from $1.03X$ to $2.3X$, and the geometric mean of the speedups was $1.34X$.

We also performed experiments with other values of the parameter K. The behavior was similar to the `LRU-2` and `SLRU-2` methods and these results are omitted in the interest of space. Our results show that: (1) the adaptive algorithms gracefully adapt over time to produce the best performance results, and (2) the performance achieved is close to the one achieved by a hypothetical offline algorithm that prefetches the data needed by each query.

5.2.3 Hotset experiment

The goal of this experiment is to show which algorithms are able to correctly identify the workload's *hotset*, and how performance is affected. Our evaluation compares the various caching algorithms with the `HotSet Prefetcher`, an algorithm that has *a priori* knowledge of the entire workload, prefetches and caches the *hotset* of partitions.

The TPC-DS like queries that we used in the previous experiment access a wide range of data that keeps evolving over time making it difficult to identify the workload's hotset, and use the `HotSet Prefetcher` to upper-bound the performance. [5] For this reason, we created a workload that operates on the 1TB `store_sales` TPC-DS fact table, and has a clear hotset. In this way, we can

[5]This is the reason we use the the per-query `Prophetic Prefetcher` to upper-bound performance of the TPC-DS like workload.

evaluate which caching algorithms are able to identify this hotset.

Our workload consists of 50 queries that contain selections, projections and aggregations. We have observed that corporate users of Big SQL and Hadoop tend to frequently access their recent data, and more rarely their older/historical data, while creating summaries for reports. Thus, the workload's hotset consists of the 250 most recently created partitions. Each query in our workload accesses a subset of the table's partitions. A partition is accessed either from the most recent 250 partitions uniformly at random with probability 0.5 (*hotset*), or uniformly from the set of the remaining 1550 older partitions (*coldset*). The total size of the 250 most frequently accessed partitions is approximately 170GB. We used a 170GB HDFS cache so that the *hotset* fits entirely in the cache.

Figure 9 shows the performance of the algorithms that we tested. The chart plots the geometric mean of the running times of the algorithms normalized to the running time using the `HotSet Prefetcher`. As shown in the figure, the `EXD`(10^{-12}) algorithm provided almost the same performance as the `HotSet Prefetcher`. This is expected as this workload is essentially the best use-case for this algorithm, which gives emphasis on the frequency of the data accesses as presented in our simulation study. However, other values of a produce different (worse) performance (e.g, `EXD`(0.3)). The parameter-free, adaptive methods were able to achieve about 95% of the performance of the `HotSet Prefetcher`.

The total elapsed time of the workload with the `Adaptive EXD` method was about 615 seconds, while the total elapsed time with the offline `HotSet Prefetcher` was 549 seconds. Note that the adaptive algorithms occasionally re-evaluate the parameter space, and thus, pay some exploration cost. Nevertheless, they are able to perform very well under various workload patterns.

Another interesting point is that some algorithms like `LRU-1` and `EXD`(0.3) resulted in higher total elapsed time for this workload (934 seconds and 885 seconds respectively) than a system that does not use the HDFS cache at all (837 seconds). The reason is that these algorithms perform multiple cache insertions that compete for resources with the query engine, essentially slowing down the workload. Setting an algorithmic parameter incorrectly can result in unexpected system behavior. When comparing the `Adaptive EXD` algorithm with the `LRU-1` algorithm, we observe that all but seven of the individual queries experienced speedups ranging from 1.08X to 6.02X, and the geometric mean of the speedups was 1.44X. This result shows that not all caching algorithms are suitable for external caches, and highlights the need for parameter-free adaptive algorithms.

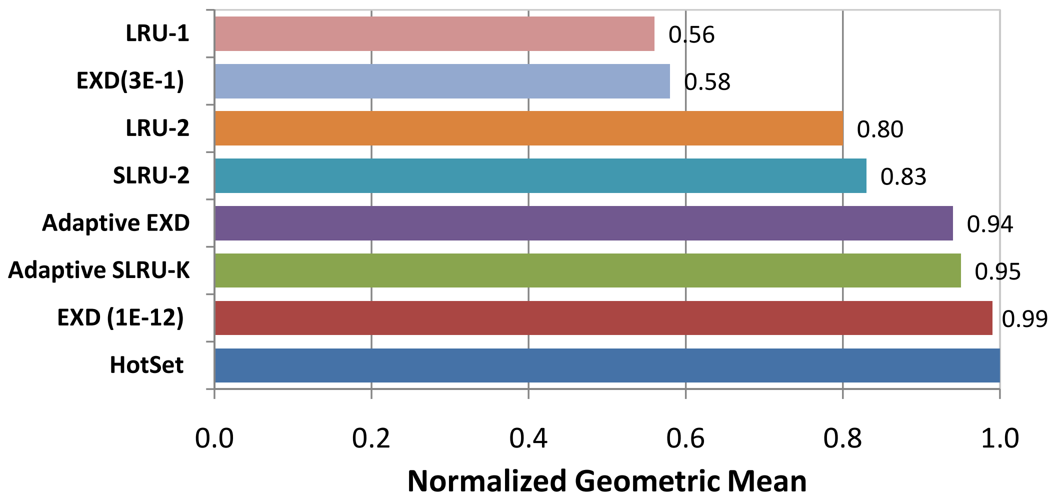

Figure 9: Comparison of various caching algorithms using the synthetic workload

5.2.4 Concurrent Workload

In this experiment, we evaluate our algorithms using a complex workload with a diverse mix of concurrent **batch** and **interactive** queries. Our goal is to investigate how the performance of interactive workloads that have low response time requirements gets affected by long running analytics workloads, such as batch queries used for reporting, running concurrently for various caching algorithms. In particular, we run batch analytics queries (the TPC-DS like workload described in Section 5.2.2) concurrently with parallel streams of interactive queries. The interactive queries are continuously executed using three parallel streams until the TPC-DS like workload finishes. We, then, evaluate how the average response time of the interactive queries gets affected by the batch queries and how the total elapsed time of the TPC-DS like workload varies with the caching method.

The interactive queries are aggregations over a single partition of a large, $1TB$ table. The table is a copy of the TPC-DS fact table used in the previous experiments (Section 5.2.3). We created a separate table for the interactive queries in order to force the batch and interactive queries to access different data sets, and thus compete more aggressively for the cache space. We used the same access pattern for the partitions of the table as in the previous experiment. More specifically, the interactive queries access a partition either from the most recent 250 partitions uniformly at random with probability 0.5, or uniformly from the set of the 1550 older partitions. Our total database size is $4TB$ and our HDFS cache size

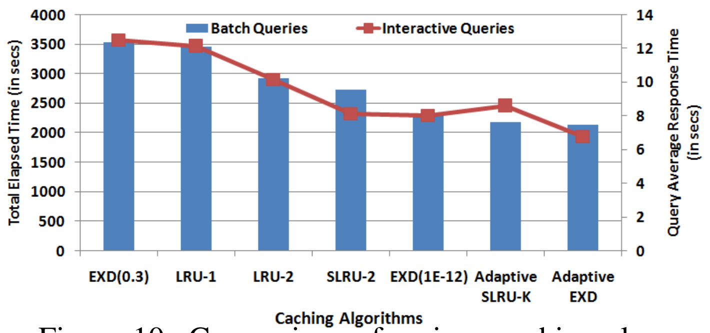

Figure 10: Comparison of various caching algorithms using the concurrent workload

is $470GB$.

To evaluate our results, we collect performance metrics for both the batch queries and the interactive queries. Figure 10 shows the total elapsed time in seconds for the TPC-DS like workload (left y-axis) as well as the average response time in seconds of the interactive queries across the three concurrent streams (right y-axis) for different caching algorithms.

As shown in the figure, the adaptive, parameter-free algorithms resulted in the lowest elapsed time for the TPC-DS like workload. Whereas the TPC-DS like workload ran for 3468 seconds with `LRU-1` algorithm, it completed in just 2145 seconds with the `Adaptive EXD` algorithm ($1.6X$ speedup). In fact, all but two of the individual queries experienced speedups ranging from $1.06X$ to $2.21X$, and the geometric mean of the speedups was $1.47X$.

Moreover, it is remarkable that the higher performance for the TPC-DS workload did not come at a cost of performance for the interactive queries. On the contrary, while the interactive queries ran for an average of 12.15 seconds using the `LRU-1` algorithm, they ran in about 6.8 seconds using the `Adaptive EXD` algorithm, an effective performance gain of $1.78X$. A similar trend was also observed for the `Adaptive SLRU-K` algorithm.

Our results show that the parameter-free, adaptive algorithms, especially the `Adaptive EXD` algorithm, can provide the best performance for both the batch queries and the interactive queries. Our findings confirm that the proposed algorithms can deliver high performance for diverse concurrent workloads.

6 Related Work

There is a lot of work in cache replacement policies developed in various contexts. For brevity, we point the reader to [23, 6] for a more comprehensive survey of the existing literature. Instead, we highlight the most closely related work to place our current work in the proper context. In the context of relational databases and storage systems, there is extensive work on page replacement policies such as the LRU-K [25], DBMIN [8], ARC [23], LIRS [17], LRFU [20], MQ [31] and 2Q [18] policies. There is also recent work on SLA-aware buffer pool algorithms for multi-tenant settings [24]. Unlike our proposed algorithms, these policies operate on fixed size pages since they mainly target traditional buffer pool settings. Moreover, these policies assume that every accessed page has to be inserted into the buffer pool, thus selective cache insertions lie beyond their remit. We also note that our algorithms focus on caching raw data, unlike approaches like semantic caching [9].

Many caching policies have been developed for web caches that operate on variable size objects. The most well-known algorithms in the space are the SIZE [2], LRU-Threshold [1], Log(Size) + LRU [1], Hyper-G [2], Lowest-Latency-First [29], Greedy-Dual-Size [6], Pitkow/Recker [2], Hybrid [29], PSS [3] and Lowest Relative Value (LRV) [27]. The work in [6] has extensively compared various web caching algorithms, and has shown that the GDS algorithm outperforms them. In this paper, we presented experiments that compare GDS with our proposed methods, and have shown that our adaptive algorithms outperform GDS.

Self-tuning and self-managing database systems have been studied in various contexts [7, 22]. In the context of caching, the ARC method [23] adapts its behavior based on the data access pattern. Unlike our algorithms, ARC operates only on fixed size objects and its adaptive design strongly depends on this assumption.

Exponential functions have been used before to model different types of behavior. For example, the work in [4] uses a power law with an exponential cuttoff to model consumer behavior in various setttings. Our proposed Adaptive EXD algorithm makes use of a parameterized exponential function to predict object re-accesses but adapts the function based on the workload access pattern. To the best of our knowledge, this is the first time that a caching algorithm makes use of an adaptive exponential function.

In the context of Hadoop systems, Cloudera [15] and Hortonworks [13], two major Hadoop distribution vendors allow the users to manually pin HDFS files, partitions or tables in the HDFS cache in order to speedup their workloads. The Impala [19] developers claim that the usage of HDFS cache can provide a $3X$

speedup on SQL-on-Hadoop workloads [15]. In the Spark ecosystem [30], Spark RDDs can be cached in Tachyon [21], a distributed in-memory file system. To the best of our knowledge, these systems do not use automatic algorithms that identify the hotset but rather rely on the user to manually cache the data.

7 Conclusions

In this work we propose online, adaptive algorithms for external caches in the context of Big Data systems. We experimentally show, through simulations and cluster experiments, that our methods are able to adjust to various workload patterns, and outperform a variety of existing static algorithms. Our experimental results show that it is essential to use an adaptive algorithm that can automatically adjust its behavior based on the workload characteristics. Because it is almost impossible to know the global system workload a priori, to identify the hotset over time, to pick the correct algorithm, and its corresponding parameter value (e.g., K, a).

References

[1] M. Abrams et al. Caching Proxies: Limitations and Potentials. Technical report, 1995.

[2] M. Abrams et al. Removal Policies in Network Caches for World-Wide Web Documents. *SIGCOMM Comput. Commun. Rev.*, 26(4), 1996.

[3] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE Trans. on Knowl. and Data Eng.*, 11(1), 1999.

[4] A. Anderson, R. Kumar, A. Tomkins, and S. Vassilvitskii. The dynamics of repeat consumption. WWW '14, 2014.

[5] Big SQL 3.0: SQL-on-Hadoop without compromise. `http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=SWW14019USEN#loaded`.

[6] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX*, 1997.

[7] S. Chaudhuri and V. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, 2007.

[8] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, 1985.

[9] S. Dar et al. Semantic data caching and replacement. VLDB, 1996.

[10] M. Y. Eltabakh et al. Eagle-eyed Elephant: Split-oriented Indexing in Hadoop. In *EDBT*, 2013.

[11] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. *PVLDB*, 7(12), 2014.

[12] Hadoop 2.0. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[13] Hortonworks: Centralized Cache Management in HDFS. `http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.1/bk_system-admin-guide/content/ch_hdfs_caching.html`.

[14] O. H. Ibarra and C. E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *J. ACM*, 22(4), 1975.

[15] HDFS Read Caching. `http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/`.

[16] K. Iwama and S. Taketomi. Removable Online Knapsack Problems. 2380:293–305, 2002.

[17] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *ACM SIGMETRICS*, 2002.

[18] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, 1994.

[19] M. Kornacker et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[20] D. Lee et al. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12), 2001.

[21] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SOCC*, 2014.

[22] S. Lightstone et al. Control Theory: a Foundational Technique for Self Managing Databases. In *ICDE Workshops*, 2007.

[23] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, 2003.

[24] V. Narasayya et al. Sharing Buffer Pool Memory in Multi-tenant Relational Database-as-a-service. *PVLDB*, 8(7), 2015.

[25] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD*, 1993.

[26] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop 's Adolescence: A Comparative Workload Analysis from Three Research Clusters, 2012.

[27] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Trans. Netw.*, 8(2), 2000.

[28] TPC-DS like Workload on Impala. `http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/`.

[29] R. P. Wooster and M. Abrams. Proxy Caching That Estimates Page Load Delays. *Computer Networks*, 29(8-13), 1997.

[30] M. Zaharia et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. NSDI, 2012.

[31] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX*, 2001.